Performance Monitoring using built in processor support in a complex real time environment

Martin Collberg
mcg01001@student.mdh.se
Erik Hugne
ehe02001@student.mdh.se

September 26, 2006

Contents

1 Abstract **5**

2 Performance Monitoring **5**
2.1 Overview . . . 5
2.2 Hardware Performance Counters . . . 6
2.3 Processor stalling . . . 7
2.4 Sampling . . . 8
2.5 Probe Effect . . . 8
2.6 Multiplexing . . . 8
2.7 Monitoring context . . . 9

3 Related work **11**
3.1 Instruction Cache Memory issues in Real-time Systems [10] . . . 11
3.1.1 Cache memory . . . 11
3.1.2 Cache memory in real time environments . . . 12
3.1.3 Performance monitoring methods . . . 13
3.1.4 Analysis . . . 13
3.2 CHUD tools - Shark [5] . . . 14
3.2.1 Overview . . . 14
3.2.2 Features . . . 16
3.3 Performance Application Programming Interface (PAPI) [1] . . . 18
3.3.1 Overview . . . 18
3.3.2 Analysis . . . 19
3.4 Digital Continuous Profiling Infrastructure (DCPI) [2] . . . 21
3.4.1 Overview . . . 21
3.4.2 Analysis . . . 22
3.5 Online Performance by Statistical Sampling of Microprocessor Performance Counters [3] . . . 23
3.5.1 Overview . . . 23
3.5.2 Analysis . . . 23
3.6 Scalable Analysis Technique for Microprocessor Performance Counter Metrics [13] . . . 24
3.6.1 Overview . . . 24
3.6.2 Analysis . . . 24
3.7 Just how accurate are performance counters? [4] . . . 25
3.7.1 Overview . . . 25
3.7.2 Analysis . . . 25
3.8 DTrace [12] . . . 27
3.8.1 Overview . . . 27
3.8.2 Analysis . . . 27

4 Problem description and method **28**
4.1 Existing Profiling tools . . . 28
4.2 Problem description . . . 28
4.2.1 Requirements Definition . . . 29
4.2.2 Sampled instruction address resolution . . . 34
4.2.3 Data flow . . . 34
4.2.4 Context switches . . . 34

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

4.3 Method 35

5 Results **36**
5.1 Time based sampling 36
5.2 Code instrumentation 36
5.3 Design of an event-driven performance monitoring tool 37
5.3.1 Overview 37
5.3.2 Daemon process 37
5.3.3 Interrupt routine 37
5.3.4 Sampler-process 37
5.3.5 Sampling 39
5.3.6 Sample-structure 39
5.3.7 Interface towards Daemon 41
5.3.8 Predefined event scenarios 42
5.4 Implementation 43
5.5 Limitations 44
5.6 Conclusions 45
5.7 Future Work 46
5.7.1 Implementing software counters for monitoring of OS behavior 46
5.7.2 Comparing profiles 46
5.7.3 Controlling measurements remotely 46
5.7.4 Call-stack trace 46
5.7.5 Graphical analysis tool 46
5.7.6 Sampler improvements 46

6 Appendix: Access configuration **49**
6.1 Mälardalen lab room 49
6.2 Network 49
6.3 User accounts 50
6.4 Services 50
6.5 Terminal server configuration 50
6.6 Node configuration 51

7 Appendix: Design of a time-driven performance monitoring tool **52**
7.1 System overview 53
7.1.1 Disk usage 53
7.1.2 CPU overhead 54
7.2 Command Line Tool 55
7.2.1 Use-Cases and State-Charts 55
7.3 Daemon 57
7.3.1 Compound statistics 60
7.4 Sampler 62
7.4.1 Overview 62
7.4.2 Multiplexing 62
7.4.3 Sampling Context 63
7.4.4 Interface towards daemon 63

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

8 Appendix: Design of an instrumentation performance monitoring tool 65
8.1 Kernel extension 65
8.1.1 Monitoring Context 65
8.1.2 Storing samples 65
8.2 HPC multiplexing 65
8.3 Public interface 66
8.3.1 System calls 66

9 Appendix: CPPMon shell commands 67

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

1 Abstract

Ericsson have expressed an interest in hardware-near profiling, using built-in performance counters in the CPU. Most boards in the Ericsson CPP platform builds upon the PowerPC processor, which have several hardware performance counters that can be used to improve performance characteristics of existing software. These have successfully been used on other platforms such as Apple´s Macintosh. There are possibly also unused research results regarding how to analyze the information in the most effective way. The purpose of this report is to provide an overview of performance monitoring and summarize some of the related work done in this field. Important aspects such as sampling methods, multiplexed monitoring, design issues when developing a performance monitoring facility and ways to interpret the monitored events are analyzed. Finally, three design suggestions are presented and compared. One of these where implemented for the CPP/OSE environment.

2 Performance Monitoring

2.1 Overview

In this chapter we will discuss the performance monitoring concept in general, but since our project focuses on how the PowerPC 750 processor handles performance monitoring, some parts will be biased towards this processor.

Performance monitoring is the process of gathering executional statistics from a system. There can be a number of reasons for doing this, for example finding bottlenecks in a program, optimizing cache usage, task scheduling analysis, optimization of algorithms etc. The four most commonly used methods for monitoring performance are as follows. [10] [11].

1. *Trace driven simulation*

   This form of static analysis uses a simulator, which takes an application's execution trace as input. The advantages of this form of analysis is that architectural elements such as cache size, bus bandwidth etc. can be altered in the simulator to make the analysis more flexible.

2. *Hardware monitoring*

   Pure HW monitoring can be achieved by attaching a sampling unit, typically a logic analyzer, to specific JTAG (Joint Test Action Group) pins on the processor. Some drawbacks of this approach is that not all processors support pure hardware monitoring, and JTAG bandwidth are severely limited which forces the processor to run at reduced speed. This is a non-intrusive solution for monitoring performance, but the data collected are at a very low level of abstraction, concerning I/O requests, memory latency etc.

3. *Software monitoring*

   In this type of monitoring, only software is used to record and collect information about the system. This can be done by instrumenting the target

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

code, inserting event-triggering functions at specific points and by capturing the events in a trace buffer. One drawback of this method is that the source code generally needs to be available in order to insert the instrumentation calls. Although DTrace [12] is one example of a framework that allows for dynamic insertion/removal of instrumentation probes in run-time. An alternative to instrumentation is event- or time-driven software probing which can be used to record executional statistics in a system. This can be implemented by using interrupt handlers, often triggered by overflow in some hardware performance counter. The probe method is less intrusive than target code instrumentation.

4. *Hybrid monitoring*

   Hybrid monitoring is a combination of software and hardware monitoring. This method is mainly used to reduce the impact on the target system done by software monitoring, and to provide a higher level of abstraction than hardware monitoring alone.

2.2 Hardware Performance Counters

Many processors today have a built in support for monitoring low-level hardware events. A set of HPC's (Hardware Performance Counters) are used to count these events, and some processors can also trigger an interrupt upon counter overflow. When gathering information about individual events it is important to put the acquired data into a useful context. Many different events may have to be monitored simultaneously to gain statistics of more useful nature and which is easy to interpret.

A common factor for processors that implement HPC's is that they cannot monitor all supported events simultaneously due to limitations in number of physical HPC registers and how they are wired. For example, the PowerPC 750 processor can only monitor 4 out of over 30 different events at any given time [6]. There is also limitations for what events each HPC can be configured to monitor. We will explain some of the most interesting events in more detail, and how they should be interpreted in order to detect and resolve performance problems.

- *L1 Instruction cache misses*

  The HPC that is set to count this event is incremented whenever a fetched instruction is not found in the L1 Instruction-Cache. The PowerPC 750 uses multi-dispatch out-of-order execution. In short, this means that when an instruction is fetched from memory it is split up in micro-instructions. These micro-instructions are queued, waiting to be executed on an appropriate functional unit in the CPU. IBM uses the term reservation stations for the different queues. The term 'multi-dispatch' is used to indicate that each functional unit has its own queue which is not always the case for other out-of-order execution processors. An Instruction cache-miss will prevent the processor from filling its issue-queues and may result in the processor being stalled while an instruction is being fetched from either the L2 cache, or in worst case the primary memory. Lowering the IMISS ratio can improve application performance considerably.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

- *Instruction Miss cycles*

  This event counts the number of cycles spent waiting for instruction fetches that missed the L1 cache to return from the L2 or primary memory. Used in conjunction with the instruction miss counter, it is possible to derive how many cycles the processor spent waiting for each instruction.

- *ITLB misses*

  This event counts the number of times an instruction address translation was not found in the Instruction Translation Lookaside Buffer (ITLB). This will result in an access to the pagetable in order to perform the virtual-to-physical address translation. Worth noting is that when an instruction address translation isn't found in the TLB, it does not necessarily mean that the instruction fetch will result in a cache miss. Code size and locality are the main factors that affect the ITLB miss-ratio.

- *Number of predicted branches that where taken*

  This events counts the number of branches that where correctly predicted by the Branch Processing Unit (BPU). Branch prediction in a CPU works by using short-term statistics to determine which path of instructions that is most likely to be executed and queues them in the pipeline. The prediction and pipelining mechanism is however very complex and will not be covered in this report.

- *Number of fall-through branches*

  This event counts the number of branches mispredicted by the BPU. I.e. branches that where not taken. The sum of fall-through branches and correctly predicted branches is the total number of branches issued, having a high quotient of fall-through/total branches results in unnecessary processor stalls. A high number of fall-through branches indicates that something is wrong with the compiler options, or the algorithm may be poorly optimized. [6] [2].

2.3 Processor stalling

When an instruction is fetched from memory and loaded in the processor, all necessary operands that the instruction will use must be available before it is allowed to retire (complete). Waiting for data to arrive from the bus is extremely costly in terms of CPU cycles, which have lead to the development of Out-of-order (OoO) processing [7]. This allows the processor to queue instructions which are waiting for data, and continue execution of other instructions. The processed instructions are re-ordered in the end to make it appear as they where executed in order. The OoO processing technology also use instruction pipelining to allow multiple instructions to be executed simultaneously on different functional units, this is called instruction-level parallelism and increases the effective use of CPU cycles. For example a FPU (Floating Point Unit) is a separate functional unit of a CPU. A processor is stalled when no instructions can retire in a cycle.

However, OoO processing and instruction pipelining is not a universal solution for solving the stall problem. Stalls are still likely to occur since the hardware cannot support all possible combinations of instructions in overlapped

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

execution. Instruction cache misses and instructions using results from other instructions as operands or waiting for access to memory will often cause stalls. A programmer writing user-applications have little control over the actual instruction pipelining, but the compiler can often be configured so that pipelining works better [1].

2.4 Sampling

In a performance monitoring context, sampling is the process of taking a "snapshot" of the system state at regular intervals. The two different sampling methods are time-driven and event-driven. In time-driven sampling, a piece of software (or hardware) called a probe, is hooked to a high resolution timer. When the timer expires, the system issues a timer-interrupt. The probe is activated on this interrupt and reads the current state of the system which is then stored as an event record. It then re-initializes the timer to a new value, which does not necessarily need to be the same during the entire sampling period. In event-driven sampling, a probe is hooked up to a specific event in the system, usually an overflow in a CPU register. When an overflow interrupt occurs or when the sample period expires, the probe is activated and stores the system state to an event record. The values that can be sampled differs between architectures, but typically the address of the latest instruction issued when the overflow occurred, program counter and current HPC values are sampled.

2.5 Probe Effect

It is impossible to achieve performance monitoring through software without introducing some executional overhead. When target-code instrumentation is used, a Probe Effect will occur when the additional instructions are inserted/removed. The Probe Effect originates from *Heisenberg's uncertainty principle*, applied to computer software. The Probe Effect can be seen as the difference of the system being tested, and the same system when inserted delays are removed. Typical errors introduced are synchronization errors for shared resources. Using an external software probe for monitoring performance means the cpu time must be shared with one or more processes related to monitoring, but it does not change the execution path of the application being tested, and will not give rise to a probe effect. [11]

2.6 Multiplexing

The definition of multiplexing is sending multiple signals in a single data stream, forming a complex signal. In analog data channels like radio traffic, Frequency Division Multiplexing (FDM) are commonly used as multiplexing method. Multiplexing of digital signals are usually accomplished through the use of Time Division Multiplexing (TDM). It is possible to use a TDM-like approach when monitoring performance in order to increase the number of logical HPC counters available. This is accomplished by splitting up the events that need to be monitored in groups and let each group occupy and configure the processors HPC's for a specific amount of time. This works pretty much the same way as how processes share a CPU in a round-robin fashion. For each group,

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

the HPC's are configured and sampled, the values are then linearly interpolated over the total time it takes for all other groups to complete their sampling runs. In this way its possible to overcome the problem with too few HPC's. The drawback of using multiplexed performance counters is that the resolution of monitored events will be lower, but it has been proven in *Online Performance Analysis by Statistical Sampling of Microprocessor Performance Counters* [3] that this method does yield acceptable results in most applications.

2.7 Monitoring context

The PMU (Performance Monitor Unit) in the PowerPC 750 does not differentiate between which process that generates the events being counted. However it is likely that an application developer is interested on the performance characteristics like cache misses for a specific process, and not global system behavior. It is possible to achieve a per-process monitoring context through the Machine State Register (MSR). Using SysCall (sc) or the Move To Machine State Register (MTMSR) instruction to set the MSR[PM] bit to 1 in a specific process, and thus enabling collection of performance events when it is running [6]. This requires that the process, or processes on which monitoring is of interest, must be modified and recompiled to enable the MSR[PM] bit. From the monitor probe perspective, only processes of interest are monitored, but if more than one process is selected for monitoring, the probe cannot separate the counted events for each process. This also puts requirements on the context switcher in the operating system. The state of the MSR[PM] bit must be saved to the process user area when the process is switched out, and restored again when it is switched back in. In OSE [8], it is possible to write a swap-in/out handler that stores the PMC register values to the process user area when a process is swapped out, and restores the values when it is swapped back in. This makes the PMC registers appear process specific, and performance events are registered in the context of every non-interrupt process in the system. When a regular interrupt occurs, the interrupted process will be accounted for the performance events generated during the time it takes for the interrupt to complete. No changes to the applications being monitored are needed, but additional overhead will occur since the swap-out handler needs to store the performance counters to the user area, and the next process must wait for the swap-in handler to restore its PMC values. The MSR[PM] bit is globally enabled and monitoring is active for all running processes.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

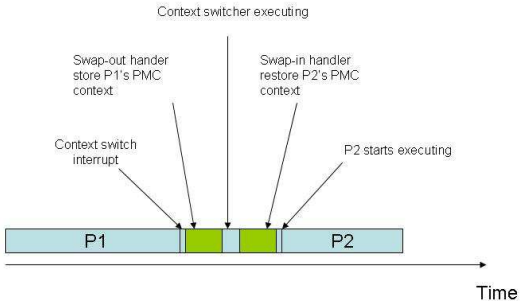

Figure 1: Swap in/out handlers in OSE

It is also possible to ignore context switches entirely, and at the time of sampling determine which process that was currently running. Since a hardware interrupt does not invoke the OS context switcher, a call to determine the currently running process will return the process ID of the process running at the time of the interrupt. A drawback of this method is that processes may be accounted for events generated by another process. This margin of error will however decrease with a higher sampling rate. Alternatively, in addition to the samples, a system relocation table can be stored in the profile. This table contains information like size, location and load-sections for all running programs. Using this, the program responsible for a taken sample can be determined offline. Programs that are started while performance are being monitored risk not being included in the relocation table. But our target environment is a rather static real-time system in which no new programs are started during normal operation. Any additions or removal of programs is accomplished through a process called system upgrade.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

3 Related work

3.1 Instruction Cache Memory issues in Real-time Systems [10]

3.1.1 Cache memory

Electronic components have doubled in capacity roughly every 18 months during the last 30 years, following Moores law. Processors now operate at such high speeds that the primary memory has problems supplying them with new instructions and data through the comparatively slow bus. A common solution to this problem is to use one or more small, fast cache memory modules on the CPU side of the bus.

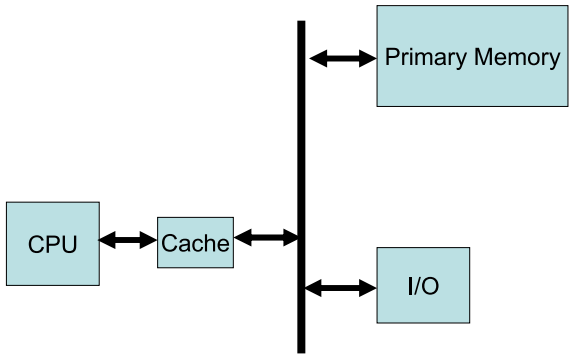

Figure 2: Simplified bus-architecture

Cache memory can be either on-chip, embedded in the CPU, or off-chip as a layer between the CPU and primary memory.

Cache memory works according to two basic principles

> *Temporal locality* (also called locality in time) concerns time. If a program is accessing an address, the chances are higher that this same address will be reused in the near future, as opposed to some arbitrary address.
>
> *Spatial locality* (also called locality in space) states that items that are close to each other in address space tend to be referred to close in time too.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

3.1.2 Cache memory in real time environments

Sebek states that you cannot guarantee a task deadline will be met in a real-time system with cache enabled. The reason for this is that the costs of refilling the cache memory after task pre-emption can be high, and difficult to measure since it depends on the intrinsic (inter-task) behavior of the pre-empting task, i.e. how much of the cached instructions for task 1 was swapped out when task 2 executed, and needs to be swapped back in when task 1 resumes.

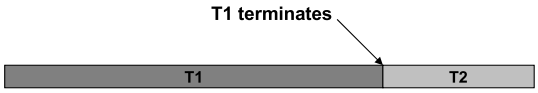

Figure 3: Two tasks execution without preemption

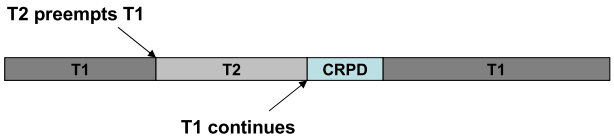

Figure 4: T2 preempting T1, resulting in CRPD

- *Extrinsic behavior* The overhead that occurs when refilling the cache when a new task is swapped in by the context switcher is called cache-related pre-emption delay (CRPD). This delay does not necessarily occur instantly after the context switch. Depending on the program design the cache-refill may come incrementally or in chunks during the execution of the task. Sebek calculates the Worst Case Execution Time (WCET) of a task with CRPD as:

$$WCET'c = WCETc + 2\delta + \gamma \quad (1)$$

  Where δ is the time needed for the operating system to make a context switch, and γ is the maximum cost for refilling the cache. However, a system using burst mode techniques for filling the instruction cache reduces the implications on execution time when refilling the cache after a pre-emption. The burst method is exploiting the spatial locality principle. Rather than fetching a single instruction from memory, an entire block of instructions are loaded.

- *Intrinsic behavior* The use of cache memory makes the execution time variable, and if the executing code generates cache-misses over a certain threshold, the code will be slower than on a system with cache disabled. This threshold is dependant on platform, architecture and operating system. Sebek presents a method to determine threshold miss-ratio and demonstrates this on the CPX2000 system. If the system is running

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

an application with a cache miss-ratio that exceeds the threshold value, cache memory should be disabled to avoid an increase in execution time.

3.1.3 Performance monitoring methods

Performance monitoring can be divided into four classes, trace driven simulation, software, hardware and hybrid monitoring.

Sebek uses a hybrid solution for measuring the CRPD on the target system. A small task called MonPoll runs at high priority, polling the performance monitor registers of the CPU and sends the data to a hardware sampler; MaMon [11]. The host system can then connect to MaMon through the parallel port and analyze the performance statistics.

3.1.4 Analysis

- *Cache memory in realtime systems*

  If an application suffers more cache misses than the threshold value, which Sebek proved to be as high as 84% on the CPX2000 system, the programmer should seriously consider profiling the code instead of, as suggested turning off the cache memory. This high threshold scenario was accomplished with synthetic code, and is extremely unlikely in a real application running on a system using burst mode transfer for filling up cache lines.

- *Performance monitoring*

  Using a hardware unit for sampling performance data is useful for making the monitoring process less intrusive. However, sampling through a software module opens up more possibilities for generating more detailed reports on the process or application-level.

- *Cache memory effects on performance*

  Sebek proves with the synthetic code used for testing cache efficiency in realtime systems that the way you write your code will affect the number of cache misses. The thesis investigates the effects on instruction cache specifically, and the results can be used to improve certain aspects of existing applications. For example aligning loops and reducing the number of cache blocks a data structure occupies during runtime.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

3.2 CHUD tools - Shark [5]

3.2.1 Overview

Shark is a tool for tuning performance of programs running on PowerPC Macintosh systems with MacOS X. It is distributed with the Computer Hardware Understanding Developer Tools (CHUD Tools) package from Apple [5].

The Shark application consists of a command-line tool and a GUI application. When performing code optimization with Shark, the first thing to do is a time profile. This is done to identify the time-intensive areas of a program. Profiles are created by sampling the running system either with the command-line tool or directly via the GUI. By specifying appropriate parameters to the command-line tool, a static analysis of object files (.o) can also be done. The profiles generated by Shark are statistical in nature, they give a representative view of what was running on the system during a sampling session. Samples can include all of the processes running on the system from both user and supervisor code.

Using the graphical application, you can study how much CPU-time each process has spent as a percentage of the total sample-time. Individual processes can be analyzed separately to see the ratio of CPU time spent within system calls, user code and interrupts. It is also possible to examine each process and its threads at source line level. The user can see execution time for each line of code both in percent of total sample-time and in seconds. Time consuming lines of code is presented in deeper shades of yellow. The user can click on a button near each statement/instruction to see advices on how to improve the performance of the code.

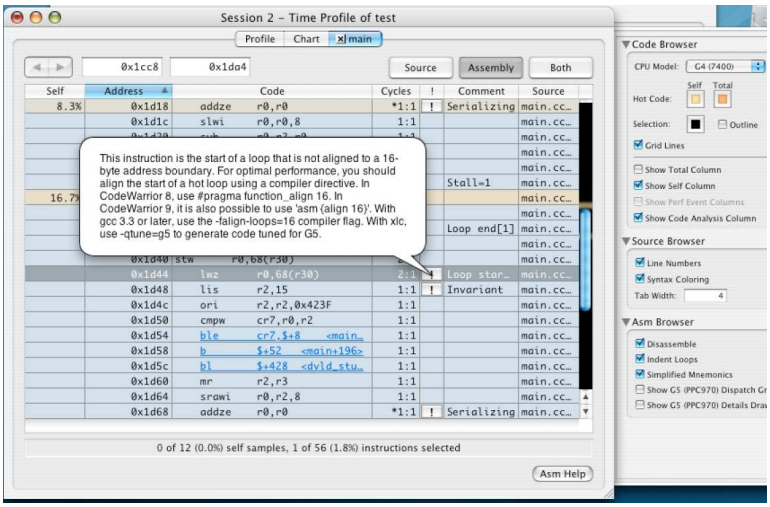
Session 2 – Time Profile of test
Profile
Chart
main
Source
Assembly
Both
Self
Address
Code
Cycles
Comment
Source
This instruction is the start of a loop that is not aligned to a 16-byte address boundary. For optimal performance, you should align the start of a hot loop using a compiler directive. In CodeWarrior 8, use #pragma function_align 16. In CodeWarrior 9, it is also possible to use 'asm (align 16)'. With gcc 3.3 or later, use the -falign-loops=16 compiler flag. With xlc, use -qtune=g5 to generate code tuned for G5.
0 of 12 (0.0%) self samples, 1 of 56 (1.8%) instructions selected
Asm Help
Code Browser
Source Browser
Asm Browser

Figure 5: Disassembled view of a time-profile

It's possible to view mixed source and assembly code if the program is compiled with debug symbols (-g with gcc). Help-sections are available for assembler instructions by selecting and instruction (line) and clicking on 'Asm help'.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

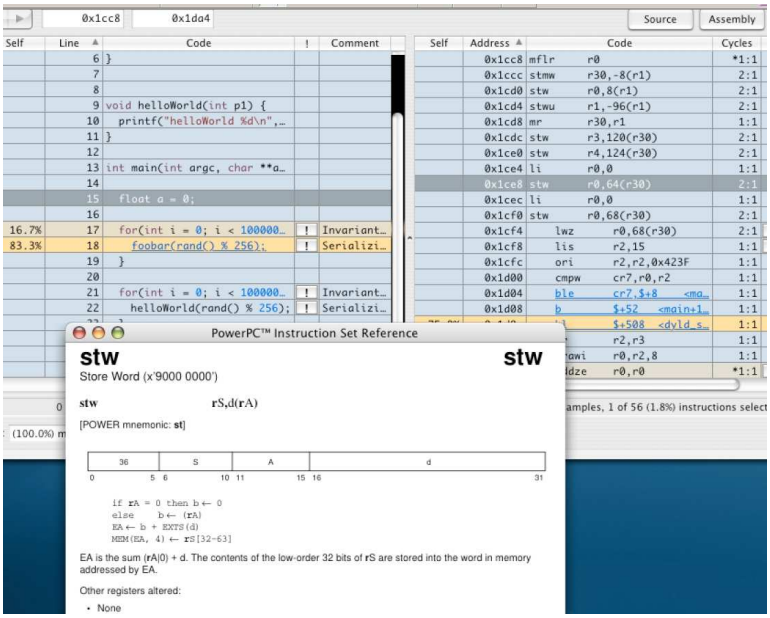
Source
Assembly
PowerPC™ Instruction Set Reference
stw
Store Word (x'9000 0000')
stw
rS,d(rA)
[POWER mnemonic: st]
EA is the sum (rA|0) + d. The contents of the low-order 32 bits of rS are stored into the word in memory addressed by EA.
Other registers altered:
None

Figure 6: Mixed source/assembly of a debug-executable

Shark works by periodically interrupting each processor in the system and sampling the currently running process, thread and instruction address. Different software and hardware performance counters are also recorded. The procedure is completely non-intrusive (the code being profiled does not need to be instrumented).

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

3.2.2 Features

Shark features and what is assumed to be required for each feature.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

Table 1: Shark features.

Feature	Description	Requirements
Source line level profiling	Shows execution time alongside lines of code.	Matching of addresses in executable code to source-lines, code compiled with debug symbols. Cycles per instruction measurements.
Process CPU Usage	Display the CPU usage ratio for different running processes.	Information about which process is currently running when a sample is taken. Monitoring context need to be switched whenever the operating system preempt a process or starts a new.
Process activity analysis	Keep track of what each process does with its time-slices. A process may perform cycles in kernel (via system calls) or user mode.	Some way of knowing if the process runs system calls or in user-code.
Remote profiling	Create a profile for a running remote target system which can be analyzed on a host computer.	Communication between target and host.
Tuning tips. Note: The tuning advices given by Shark are mostly static and often concerns performance problems that could be resolved by specifying the correct flags at compile-time. Shark gather the information needed to provide tuning tips by analyzing code, and not how it runs.	Present advice on how to solve different problems related to performance.	Good knowledge of the microprocessor's behavior. Static analysis of compiled code.
Performance event counting	Collect data from hardware & software performance counters and display the results in graphs.	A kernel extension (module) that handles the configuration and reading of the counters. Graphical representation of the results.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

3.3 Performance Application Programming Interface (PAPI) [1]

3.3.1 Overview

The developers behind the PAPI project is trying to define a standard for accessing the hardware performance counters present on many CPU's today. It provides a set of interfaces that the developer can implement in their application to measure performance events at specific locations in the target code. Two user-level interfaces are provided for performing performance measurements. One high-level through which basic events common to the RISC processors can be counted, and a low-level interface that can be used to count machine-specific events. Statistics derived from a combination of performance events can sometimes prove to be more useful than the counter values alone.
For example:

- *Level 1 Cache hit-ratio*

  $$\alpha = 1.0 - (\beta/(\gamma + \delta)) \tag{2}$$

  Where α spans between $0 - 1$ indicating the ratio of successful L1 cache accesses. β indicates the total number of L1 cache misses, γ is the number of completed load instructions and δ represents the number of completed store instructions.

- *Level 2 Cache hit-ratio*

  $$\eta = 1.0 - (\epsilon/\beta) \tag{3}$$

  η spans between $0-1$ and indicates the ratio of memory accesses missing the L1, but hitting the L2 cache. ϵ is the total number of L2 cache misses. High values of α or η, indicates good L1 or L2 cache performance.

- *Completed operations per cycle*

  $$\sigma = \omega/\theta \tag{4}$$

  σ is a fractional value indicating the total number of operations issued per cycle. A low value of σ indicates frequent processor stalls possibly due to an inefficient program. ω is the total number of instructions completed and θ is total CPU cycles.

- *Memory access density*

  $$\lambda = (\gamma + \delta)/\theta \tag{5}$$

  High memory access density, λ, does not necessarily indicate inefficient code, but it will have a negative impact on performance.

Papi is constructed in a layered design to make it as portable as possible. It is divided into two main parts, one machine-independent that handles states, memory management, manipulation of data-structures, and everything that doesn't have a direct coupling to the underlying architecture. This layer can also emulate some of the more advanced features such as overflow handling even if it is not natively supported by the OS/hardware. The other, machine-dependent layer contains the methods for accessing and initializing the hardware counters.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

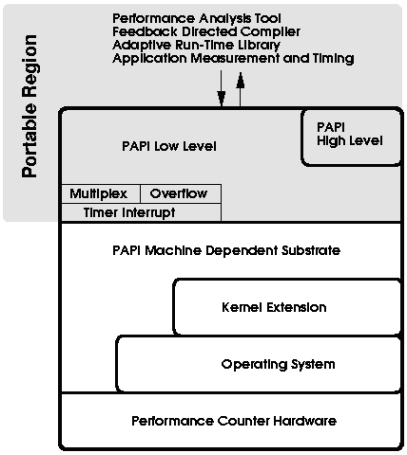

Figure 7: Architecture of PAPI

3.3.2 Analysis

The compound statistics defined in PAPI are derived from basic RISC events and can be implemented on any RISC processor, for example the PowerPC 750. If the processor does not have native support for counting all events simultaneously, a HPC event multiplexing method can be used. The software multiplexing functionality is implemented in the portable region of PAPI, I.e. the process of multiplexing hardware counters is done in user-space, which means that a kernel boundary crossing is necessary whenever a new set of events is scheduled for monitoring. The transition between user/kernel-code is a time-consuming process, which could be avoided if multiplexing where to be done entirely in kernel-space. Moreover, PAPI is designed around instrumentation of the target code, which means that the developer must embed calls

in the target code to one of the API's for initializing, starting and stopping the performance counters. This intrusive form of monitoring performance should be avoided. It may be possible to create a software probe which implements the behavior for initializing, starting and stopping performance measurement. This solution puts some requirements on how the operating system handles context switches, and we will try to determine if it is possible during our analysis. PAPI have existed for several years, and during this time a number of front-ends for displaying event statistics have been created. For example Perfometer and Profometer. And also some more advanced profiling tools like Visual Profiler, SvPablo and DEEP. Using PAPI should allow for further extensions to, for example control the probe from a host computer.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

3.4 Digital Continuous Profiling Infrastructure (DCPI) [2]

3.4.1 Overview

DCPI is a profiling tool aimed for continuous monitoring of production systems. The key aspects are low overhead and high sampling rate. DCPI is able to classify processor stalls from sampling the program-counter (PC). The performance data is collected using the non-intrusive software probe method, sampling at a system-wide level in random time intervals. The number of samples collected at each instruction address (PC value) is proportional to the total time spent executing that instruction. DCPI also allows for monitoring of system events such as cache-misses if the processor supports it. DCPI contains a number of analysis tools to generate histograms, showing execution time spent per image, procedure, source line and instruction. More advanced analysis tools also exist for analyzing processor stalls, and annotating source code with possible explanations for these (dcpicalc). This information is deduced from the sampled performance data in conjunction with the executable image.

```
for(i = 0; i < n; i++) {
    c[i] = a[i];
}
```

```
***Best-case 8/13 = 0.62 CPI
**Actual 140/13 = 10.77 CPI

Addr        Instruction                    Samples   CPI Culprit
            pD (p = branch mispredict)
            pD (D = DTB miss)
009810      ldq     t4,0(t1)               3126      2.0cy
009814      addq    t0,0x4,t0              0         (dual issue)
009818      ldq     t5,8(t1)               1636      1.0cy
00981c      ldq     t6,16(t1)              390       0.5cy
009820      ldq     a0,24(t1)              1482      1.0cy
009824      lda     t1,32(t1)              0         (dual issue)
        dwD (d = D-cache miss)
        dwD ... 18.0cy
        dwD (w = write-buffer overflow)
009828      stq     t4,0(t2)               27766     18.0cy 9810
00982c      cmpult  t0,v0,t4               0         (dual issue)
009830      stq     t5,8(t2)               1493      1.0cy
        s   (s = slotting hazard)
        dwD
        dwD ... 114.5cy
        dwD
009834      stq     t6,16(t2)              174727    114.5cy 981c
        s
009838      stq     a0,24(t2)              1548      1.0cy
00983c      lda     t2,32(t2)              0         (dual issue)
009840      bne     t4,0x009810            1586      1.0cy
```

Figure 8: DCPIcalc output

Since DCPI is designed for continuous profiling, careful design decisions have been made regarding the data collection system to minimize the CPU overhead, disk and memory usage. The system consists of three major components, the kernel device driver, which handles HPC interrupts and aggregate the samples in a hash table by counting the number of times a specific event have occurred, at a specific address in a specific program. The daemon process extracts the sampled data from the device driver and stores these in a profile database. A modified system loader associates running processes with their executable image file.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

3.4.2 Analysis

Multiplexed sampling have not been considered in DCPI, a likely reason for this are that it will have a negative effect when used in continuous profiling in a production system. The storage needs for the kernel driver buffer and user level daemon database will be higher, and the additional overhead for switching monitored event and extracting the data from the kernel driver will have a considerable impact on execution time.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

3.5 Online Performance by Statistical Sampling of Microprocessor Performance Counters [3]

3.5.1 Overview

In this article, multiplexing is presented as a method for increasing the number of logical HPC's. The reason for doing this is that most microprocessors provide more measurable events than HPC's available to measure them simultaneously.

3.5.2 Analysis

The performance monitor facility used in the article, has been implemented in two functional modules. One module works within the kernel and is responsible for configuring the PMU (Performance Monitor Unit), handle multiplexing and reading HPC counters. There is also a programming interface available as a user-level library for communicating with the kernel-module. Placing all multiplexing functionality within the kernel is important to reduce the overhead when switching monitored events. A different approach is taken with PAPI [1], where kernel-boundary crossings occur every time a new set of events need to be measured. The reason for doing this is to improve platform independence.

3.6 Scalable Analysis Technique for Microprocessor Performance Counter Metrics [13]

3.6.1 Overview

In this article different statistical techniques are discussed for extracting useful information from the the data acquired during the sampling of HPC's. When a large set of events are monitored over longer periods of time, the vast number of datapoints generated can easily eclipse the important characteristics of the data. The article focuses on techniques such as Clustering, Principal Component Analysis (PCA) and factor analysis with covariance matrices to isolate interesting properties of a data set.

3.6.2 Analysis

The techniques described in the article are useful when analyzing and presenting data from a profile. For example, different visualization techniques in a graphical analysis tool could be based on clustering or PCA. The article is focused on mathematical solutions for improving the usefulness of measured data.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

3.7 Just how accurate are performance counters? [4]

3.7.1 Overview

Common to processor architectures are that they seldom, if ever, provide any documentation on how accurate the HPC's are. This study presents a methodology for determining the accuracy of the HPC events that are reasonably predictable. Three microbenchmarks are defined;

- *Linear Microbenchmark*
  This test is designed to measure the L1 I-cache miss event specifically, and to try to ascertain how accurate the event counter actually is. A repeated sequence of add instructions is used in the test, and no branches are used in order to avoid speculative execution.
- *Loop Microbenchmark*
  In this test, the number of decoded instructions, load/store events and resolved conditional branches are measured. The test code is similar to the Linear Microbenchmark, encapsulated in a for-loop.
- *Array Microbenchmark*
  This test measures the number of L1 D-cache, L2 cache and TLB miss events. The test code is displayed below.

```
#define MAXSIZE 1000000
int main(int argc, char *argv[]) {
    int a[MAXSIZE], ARRAYSIZE, i;
    ARRAYSIZE = atoi(argv[1]);
    for(i = 0; i < ARRAYSIZE; i++) {
        a[i] = a[i] + 1;
    }
}
```

The predictions are done using parameters relating to the architecture (MIPS R12000) like cache memory size, block-size and page size, in event-specific formulas. The performance measurements are accomplished through the use of Perfex,which consists of two modules. libPerfex, a library of C/Fortran functions that the programmer can use to initiate and stop measurements at specific code sections inside the target application. Perfex is a command line tool that can count events for an entire executable image.

The tests are performed on a MIPS R12000 simulator, and the accuracy is defined as the quotient of measured events and predicted events. Common to the three microbenchmarks are that measurements accomplished through instrumentation of code (libPerfex) are more accurate than application-wide measurements (Perfex), The study shows that the accuracy of performance measurements will increase with the number of instructions executed, I.e. measurement time.

3.7.2 Analysis

This report proves that code instrumentation will yield more accurate measurements. However this type of performance monitoring is time-consuming

for the developer. Good knowledge of the architecture is also needed in order to achieve meaningful results. External monitoring through the use of a software probe relieves the programmer from this, and a performance analysis can be performed by another developer without having access to- or knowledge of the source code.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

3.8 DTrace [12]

3.8.1 Overview

DTrace is a built-in tool in Solaris that allows for tracing of both user-programs and OS behavior. The trace-functionality is accomplished through the use of small software probes, written in the D script language. The DTrace framework resides in kernel-space and provides functionality such as data-buffering and processing of the probes. A set of loadable kernel modules, called *Providers* is responsible for runtime-insertion of the compiled probes at appropriate locations. When a new probe have been defined and registered with a provider, any process can use them through the DTrace framework API. These processes are called *Consumers* and are responsible for extracting the buffered data from the framework.

```
probe description
/predicate/ {
   actions
}
```

Probe description specifies when and where instrumentation should be used. For example *proc:::exec-success* means that the script will be run whenever a new process was started in the system.
The *Predicate* puts further restrictions on when the D script should be run. For example, the predicate *cpu = 0* limits the script to only run when new processes have been started on cpu with id 0.
Actions specify what should be done when the event occurs. For example:

```
printf("%s(pid=%d) started by uid - %d\n",execname, pid, uid);
```

3.8.2 Analysis

DTrace claims to have a ”zero probe effect” when the probes are disabled. However the major drawbacks of instrumentation is that the difference in execution time introduced when the probe is inserted/removed can change the applications behavior. Typical errors introduced are synchronization errors when processes attempt to access the same resource which can lead to critical race-conditions.

El Shobaki presents three different methods for eliminating the probe-effect [11].

1. *Leaving the probes in the final system*
   Which can have a considerable impact on the applications performance.

2. *Include probe-delays in schedulability analysis*
   This method does not guarantee the ordering of events, and unforeseen synchronization errors is still a risk.

3. *Use non-intrusive hardware.*
   DTrace relies on the software instrumentation probes, and a hardware solution is not viable. Neither will a hybrid solution for sampling the data change the fact that inserted/removed instrumentation will introduce a probe-effect.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

4 Problem description and method

4.1 Existing Profiling tools

There is an existing application for measuring performance in the OSE/CPP environment, called PerfMon. This application features counting of events within the CPU (PowerPC 750). Four events can be counted simultaneously. However counting is about the only thing that PerfMon does. There is no multiplexing, no storing of profiles or coupling of events back to source-code. These limitations leaves the user (application developer), with a vague picture of how changes in code affects performance.

It is possible to see that the counts of a certain event has decreased or increased between different runs, but there is no way to determine which parts of a software project that is in the need for further improvement. Which leaves the responsibility to the programmer to know where bottlenecks is most likely to occur.

4.2 Problem description

The problem addressed in this thesis is to provide Ericsson AB with a way to measure performance on their PowerPC 750 based general purpose boards used in their Connectivity Packet Platform.

Our goal is to analyze how to use the performance monitoring facilities within the PowerPC 750 to extract useful data from the CPU and store this data into a profile. Accessible to a software developer.

Our system is going to focus primarily on the following tasks:

1. Select which events to sample.

2. Sampling of CPU registers.

3. Save sampled data to disk into a profile, which can be used for further analysis.

4. Couple events back to source-code.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

4.2.1 Requirements Definition

Table 2: Requirement Sources

Source	Description	Remark
Stud	Erik Hugne & Martin Collberg	
Sup	Daniel Flemström & Jukka Mäki-Turja	
Cust	Ericsson AB	

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

Table 3: Requirements Definition

ID	Status	Priority	Description	Source
C1	I	10	**Command line tool** *Definition:* Utility for parsing user input and take the appropriate action. *Motivation:* A user interface for controlling the monitoring process is required. Through the command-line tool the user will be able to: • specify sample-rate (which will remain the same throughout the session. • specify which events that will be monitored (including compound statistics). • specify sampling time. • stop an ongoing sampling. • specify where profile should be stored.	Stud

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

D1	I	9	**User-level Daemon** *Definition:* Process that runs with user-level privileges and communicates with the sampler. *Motivation:* Adding a layer of abstraction between the hardware dependent sampler and the user interface, eases future implementation for other processors than the PowerPC 750. The daemon will be unaware of the underlying processor specific sampler implementation.	Stud
D2	I	8	**Daemon interface** *Definition:* A well defined interface that allows for future integration of user-end components. *Motivation:* It is likely that a graphical representation of profiling results on a remote workstation will be needed. The interface should also allow for controlling the daemon remotely (start, stop, receive results).	Stud
D3	I	8	**Daemon configuration** *Definition:* A configuration file that specifies the processor type, events available and definitions of compound statistics. The configuration will define these compound statistics via a simple script that is parsed by the daemon. *Motivation:* The performance monitor tool should be usable for many different processors. Using a configuration file for each processor will	Stud
DS1	I	10	**Profile storage** *Definition:* Storage of a profile in memory, which can be saved to disk later. *Motivation:* The profile needs to be stored continuously during the sampling. A profile is saved to disk on the target once the sampling is complete or the user chooses to stop the sampler via the command-line tool.	Stud

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

DS2	I	9	**Signal Handler** *Definition:* Communications module for kernel driver - daemon process using OSE signals. *Motivation:* A means for the kernel driver and daemon process to communicate is needed.	Stud
P1	I	10	**No instrumentation of target program** *Definition:* Monitoring is accomplished through a software probe. No instrumentation of the code that is being analyzed will be needed. *Motivation:* The reasons for this requirement is that the probe-effect that occurs when instrumenting code can cause unpredictable behavior of the target program and it puts additional workload on the application developer.	Stud
P2	I	10	**Low performance monitoring overhead** *Definition:* Monitoring should have a low impact on performance and not interfere with running processes. *Motivation:* The goal of performance monitoring is to find out how applications behaves in a production system, this will be compromised if the monitoring process have a high resource consumption.	Stud
P3	I	8	**Compound statistics** *Definition:* Combining raw HPC events in order to obtain more useful statistics. *Motivation:* Relations between different hardware events are often more useful than raw measurements.	Stud
S1	I	8	**Multiplexing of HPC's** *Definition:* Increasing the number of logical HPC's through TDM. *Motivation:* The are four physical HPC's in the PowerPC 750. Using multiplexing will enable us to increase the number of simultaneously measurable events at the cost of lower sampling resolution.	Stud

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

S2	I	9	**Variable sampling rate** *Definition:* Ability to specify the sampling rate when starting a monitoring session. *Motivation:* Depending on the application and number of events being measured, the optimal sampling rate will be different.	Stud
S3	I	7	**Sampling context** *Definition:* A facility for determining which process should be charged for the sampled events. *Motivation:* By charging samples to separate processes, per-process statistics can be obtained.	Stud
S4	I	7	**Sampling of instruction address** *Definition:* When a sample is taken, the address of the last instruction retired is sampled. *Motivation:* Sampling the instruction address when an interrupt occur will allow us to associate the sampled events with lines of code (With more or less accuracy depending on the sample-rate used).	Stud
S5	I	10	**PowerPC 750 specific sampler** *Definition:* A sampler which implements reading and multiplexing of HPCs and initialization of registers associated with the PMU of the PowerPC 750. *Motivation:* Isolating all the processor specific functionality in one software module will increase portability. Additional sampler implementations will allow for supporting other types of processors.	Stud

Requirement status:

- *I = initial (this requirement has been identified at the beginning of the project),*
- *D = dropped (this requirement has been deleted from the requirement definitions),*
- *H = on hold (decision to be implemented or dropped will be made later),*
- *A = additional (this requirement was introduced during the project course).*

Priority:
10 = highest, 1 = Lowest.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

4.2.2 Sampled instruction address resolution

In section 2.4, we discussed two approaches for collecting samples from a running program, Time driven and Event driven sampling. Time driven sampling can provide a good overview for how an application is performing. When sampling many different events, its possible to determine the cause of an increase in CPI (clocks per instruction) by correlating CPI spikes with the other simultaneously sampled events. One problem with this approach is that its harder to tie the events being sampled to an address of execution. Knowing this address is necessary in order to refer back to the source code and pinpoint the function or instructions causing the performance drop. As an example, a sampling interval of 1ms on a 750MHz processor will result in each sample having a span of 750 000 cycles. The accuracy of sampled instruction addresses when using this approach is relatively low compared to event driven sampling, where the problem becomes less apparent since samples are collected at, or in close proximity to where the events occur.

4.2.3 Data flow

The occurrence frequency of events depends on the executing program code and the type of event. The size of a profile will grow linearly when events are collected at fixed time intervals, but is harder to predict in an event driven solution. Typically cycles-related events like level 1 cache miss-cycles occur at a much higher rate than the level 1 cache misses that cause the cache miss-cycles counter to be incremented. This must be taken into consideration when selecting the threshold for how many events that are allowed to occur before the HPC values are sampled.

4.2.4 Context switches

We stated earlier in our analysis of PAPI, section 3.3.2 the problem of handling context switches when monitoring performance in a multitasking operating system. The OSE program handler (*prh*) provides a signaling interface for accessing the program relocation table *(PrhListProgramsVerbose)*. This table includes program name and version, the size of the program and where it is loaded in memory. If this table is included in a performance profile, it would be possible to tie each sample to a specific program when the profile is processed on a host machine. This means that context switches can be ignored during the performance monitoring process, resulting in a smaller and faster sampler.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

4.3 Method

The existing tool at Ericsson called PerfMon did some rudimentary measurements using HPC's. But it lacked the ability to store this information in a reuseable way. And also to tie performance drops to specific regions of code.
In our solution we determined that performance measurements *(profiles)* need to be stored in a non-volatile memory in order to be able to look back on previous measurements and compare the results. In addition to verifying if the profiled code actually resulted in a performance increase, it is also possible to determine weather the introduced changes have caused any performance problems elsewhere.
Achieving good performance can either be a ongoing task during development of a system where strict requirements are set at the early stages of a project.

In many cases, especially in real-time systems, application and system performance may be of lower priority than for example product stability, reliability and time to market. However, performance may need to be analyzed after the product has been established to further satisfy the customers and remain in competitive advantage.

From our analysis of related work and existing implementations, we have created three design suggestions.

1. A time-driven solution, similar to Shark and DCPI where HPC values are sampled at a certain interval and stored to file. The HPC multiplexing feature presented in this design makes it possible to statistically determine the cause of a performance drop as explained in Multiplexing, section 2.6.

2. An event-driven solution that focuses on target code instrumentation, inspired by PAPI, section 3.3.1. The developer have a high degree of control over the measurements, but it is complex to use and cannot monitor global system behavior.

3. An event-driven solution that relies on a software probe to sample the HPC values. This solution provides a higher SIA resolution, section 4.2.2, than time-based sampling, and the cause of a performance drop can be identified in the code through the sampled addresses.

Complete design descriptions of these can be found in appendices 7, 8 and 5.3.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

5 Results

From our three initial design suggestions, the event-driven solution was selected to be implemented in the OSE/CPP environment, the design details are described thoroughly in section 5.3. In this section, we provide a summary of the two suggestions not implemented, complete design descriptions of these can be found in appendices 7 and 8. Our solution addresses most of the problems with the existing PerfMon application discussed in section 4.1, but it does not provide multiplexing of HPCs.

5.1 Time based sampling

The time driven solution builds on a daemon process that runs in user space, serving as a user interface towards a kernel driver. The kernel driver is responsible for configuring the processor registers related to performance monitoring, and to periodically sample selected registers and store this data into a buffer. The main advantage of this solution is that it facilitates the use of HPC multiplexing. This makes it possible to measure more events simultaneously with a limited set of physical HPCs, but the granularity of the samples will be high, and it is hard to tie performance issues back to the executing source code.

5.2 Code instrumentation

This solution does not build on the daemon interface towards the kernel driver. Performance samples are collected in an event-driven fashion, but the responsibility of configuring, starting and stopping measurements is put on the application programmer. This is accomplished through extending the set of available system calls in the kernel driver, allowing an application programmer to perform performance analysis during development by embedding calls to the performance monitor driver. The benefits are a high coupling to source code, but it will be harder to correlate the results between runs, and the concept of inserting these calls into production code does not appeal to the software designers.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

5.3 Design of an event-driven performance monitoring tool

In this section, we will present the design of a non-intrusive, event-driven performance monitoring application for the CPP-platform/PowerPC 750 processor. By non-intrusive we mean that no target code has to be altered in-order to measure performance. Samples are taken of the internal processor state (current execution address & HPC's) when hardware events occur. The main motivation for this approach is to get a good coupling between the source code and performance problems, and to some extent find out what events causes bottlenecks in the whole system or in some specific part of a program.

5.3.1 Overview

The design can be broken down into three main parts. A sampler that runs in supervised mode (kernel). An interrupt routine that handles the time-critical parts of the sampling process. A daemon process that runs in user-space, serving as an intermediate-layer between the user and the sampler.

5.3.2 Daemon process

The commands issued by the user to start and stop performance monitoring is handled by the daemon. When the start command is invoked, a sampling configuration is constructed from the parameters. A number of predefined scenarios will be available (see section 5.3.8), but the user will not be restricted to using them.
Since the amount of samples taken during a run can be large, the data needs to be transferred continuously from kernel resources to persistent memory. This is taken care of by the daemon running in user-space. The sampler notifies the daemon when data is ready to be retrieved. The daemon then reads from the sample-buffer using a syscall provided by the sampler and sequentially updates the profile.

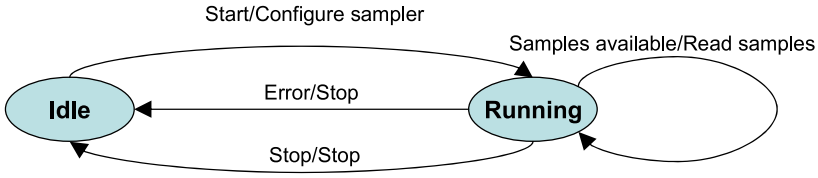

Figure 9: Statechart for the user-level daemon

5.3.3 Interrupt routine

The interrupt routine is executed when an overflow has occurred in any of the HPC's. It handles the tasks critical to the current CPU state when an event has occurred. To minimize the impact on performance of other running processes, the code for the interrupt routine needs to be kept small and effective.

5.3.4 Sampler-process

The sampler-process runs in privileged-mode. Its purpose is to handle the transferring of buffered samples to the daemon-process, limit the complexity

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

of the interrupt routine and initiate different monitoring scenarios. This process resides within the same memory boundaries as the interrupt routine (the kernel).
Samples are collected at system scope, meaning that the individual samples does not have a direct coupling to the process that generated the event. This can however be done offline when the samples are processed since we have access to the instruction address where the event occurred. Relocation information for the processes loaded on the target must be available together with the profile in order to find out which process each address belongs to.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

5.3.5 Sampling

The profiler can be configured to measure four different events simultaneously (limited by the number of physical HPCs on the PowerPC 750). Each counter is mapped to a specific event and configured as a counter or as a trigger. A trigger has a threshold parameter, so that it is possible to tune how often samples should be taken depending on the type of event. The counters will be sampled when any of the triggers causes an overflow-interrupt to occur.

5.3.6 Sample-structure

Each sample includes a 64-bit timestamp (register TBU & TBL). By including the time passed between each specific event, it is possible to determine its occurrence frequency.

```
// Header for a profile
struct profile_header_s {
    U32 EVENTSEL[4];     // Bitmask of events
                         // mapped to HPC 1-4
    U32 THRESHOLD[4];    // Threshold values for
                         // HPC 1-4
};

// Sample-structure
struct sample_s {
    U32     TBU;     // Time Base Upper register
    U32     TBL;     // Time Base Lower register
    U32     HPC[4]; // HPC 1-4 values
    U32     SIA;     // Instruction executing while
                     // the sample was taken
    UCHAR   TRIG;    // HPC that triggered
                     // sampling (bitmask).
};
```

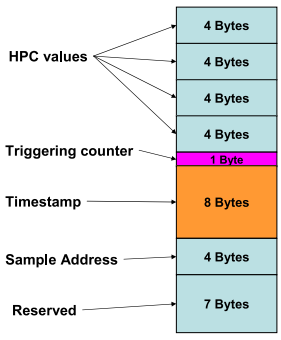

Figure 10: 64 bytes sample structure

Theoretically, a HPC configured as a counter may overflow before any of the triggers. If this happens a sample is taken but with the TRIG field set to

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

zero. The counter-values of such samples can be added together offline while parsing the profile to produce values with higher resolution than 32-bits. This limits the size of the sample-structure and since each sample is marked with a timestamp it is easy to do this offline.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

5.3.7 Interface towards Daemon

The interrupt routine stores samples continuously into a memory area shared with the sampler process. This memory will be split up in segments to allow for double-buffering. When one segment is full, the sampler will notify the daemon which will then read the data. A syscall will be implemented to let the daemon copy samples from the kernel buffer. During this data-transfer, the interrupt routine will still be able to fill the other segment with sampled data.

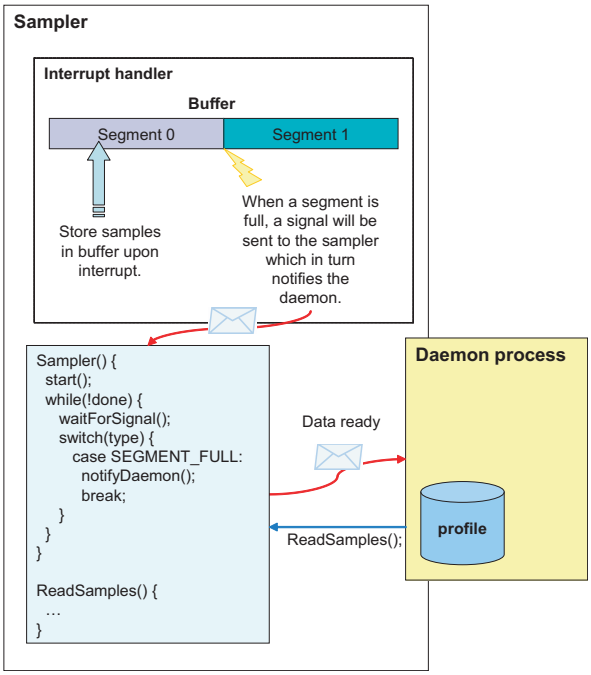

Figure 11: Communication between daemon, sampler and the interrupt routine

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

5.3.8 Predefined event scenarios

- L1 cache measurements:
  C Instructions completed, excluding folded branches.
  T L1 instruction-cache misses.
  T L1 data-cache misses.

- L2 cache measurements:
  C - Number of accesses that hit the L2 cache, including cache operations.
  T - L2 instruction-cache misses.
  T - L2 data-cache misses.
  C - Instructions completed, excluding folded branches.

- TLB measurements:
  C - Number of cycles spent performing table search operations for the ITLB.
  T - ITLB misses.
  T - DTLB misses.
  C - Number of cycles spent performing table search operations for the DTLB.

- Instructions per cycle measurements:
  C - Number of valid instruction effective addresses delivered to the memory subsystem.
  T - Instructions dispatched.
  T - Instructions completed, excluding folded branches.
  C - Processor cycles.

- Branch measurements:
  T - Branch unresolved when processed.
  T - Branch misprediction.
  C - Number of stall cycles in the branch processing unit due to LR or CR unresolved dependencies.

T = events that trigger sampling.
C = events that will be counted between samples

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

5.4 Implementation

We have implemented a tool that samples hardware performance events in the PowerPC 750 CPU using the event-driven approach presented in 5.3. The test environment we used was a general purpose board (GPB) on a CPP node running a realtime operating system from Enea, OSE Delta 4.5.1

Selected events are measured and stored in a profile on the target filesystem. Additionally, a profile contains a header, holding the sampling configuration, and a relocation table with information about all programs running when the session was started.

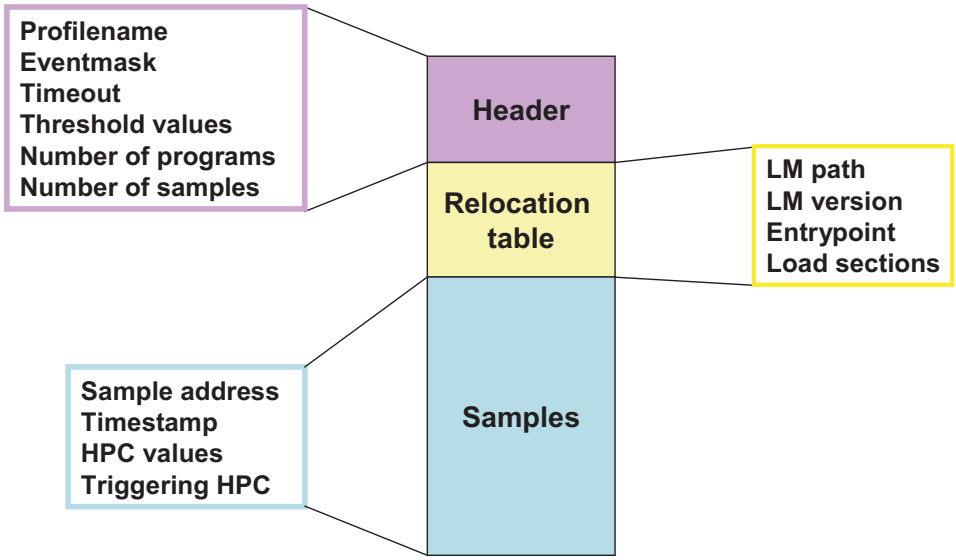

Figure 12: Profile structure

- The first 64 bytes of a profile is the profile header.
- The following X*552 bytes is the relocation table, where X is the number of programs, described in the profile header. The relocation table is fetched from the board program handler.
- The rest of the profile contains samples, each sample is 64 bytes large.

The monitoring framework is implemented as a part of the Basic operating system, and as a loadmodule running with user-privileges. Two OSE shell commands are added to communicate with the performance monitoring service and they are fully documented in appendix 9, *CPPMon shell commands*.

The *cppmon* command is used to configure, start and stop measurements, as configuration a number of parameters is given which selects the hardware events to sample and at what rates (thresholds).

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

Only four events can be sampled simultaneously due to the hardware limitations of the PowerPC 750. Its also possible to view the contents of a generated profile by specifying additional parameters.

The second command, *smpdiag*, is only used for viewing PMU related registers and see diagnostic values reported during a session regarding buffers and sampler state. This is useful when performing custom measurements and experimenting with different threshold values.

5.5 Limitations

Our project is limited to the actual sampling of HPC events and to store samples taken into a profile. Means for analyzing the profile is out of scope for this project. It is however necessary to implement an analysis tool for the profiles in-order to make the framework useful in a production system. Such a tool could display graphs of sampled events and provide an easy way to couple events to sourcecode using the information stored in the profile. Maybe clicking on a graph curve will take you to the section of code responsible for a set of events. We have created a rudimentary graph analyzer in Eclipse TPTP with this type of functionality. It includes a log parser for Excel CSV files, but since it cannot in it´s present state analyze the profiles generated by CPPMon, we have chosen not to describe this application in further detail.

There is no guarantee that all samples taken can be buffered and stored to disk. The amount of data needed to be written to disk is affected by three parameters, the threshold values of each counter, the type of events measured and the behavior of the running programs.
Different events occur more or less often and its hard to predict how many events that will occur within a given time-frame. This causes problems when trying to predict I/O bandwidth usage.
The monitoring framework is failsafe in such a way that overflowing a buffer will not cause severe application failure or node restart, but rather only loss of samples. This is only likely to occur when performing custom measurements, and we provide the smpdiag tool to assist in creating custom measurement configurations.

One serious problem that we have not addressed arises when setting the thresholds to extremely low values (like trigger sampling on every instruction completed) if such settings are used, the watchdog in the basic OS will bite and the node will restart. We assume that this is due to the interrupt being taken too often, so that the CPU is not able to perform other tasks like kicking the watchdog (resetting the watchdog timer). We propose two different solutions to this problem in future work section 5.7.6

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

5.6 Conclusions

The CPPMon framework provides a simple interface for accessing the PowerPC 750 performance monitoring unit. Measurements are stored in a structured binary profile that can be processed offline. The framework is configurable and should be useful in many scenarios where performance events need to be monitored.

Since CppMon is working through the external probe concept, no instrumentation of code is necessary. This should make CppMon useful both during application development and after a product release.

The profiles generated can be used for analyzing performance characteristics of a program or of the system as a whole. Combined with the relocation information sampled events can be tied back to sourcecode.

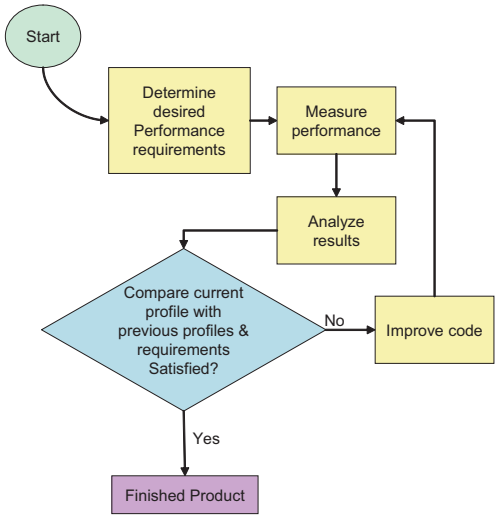

Figure 13: Profiling Work flow

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

5.7 Future Work

In this section, we will present ideas for future work related to performance monitoring.

5.7.1 Implementing software counters for monitoring of OS behavior

Some system events cannot be monitored using hardware counters alone. For example, poor cache performance could be a result of the CRPD caused by high frequent context switches. Adding a software counter for monitoring context switches will make it possible to determine this.

5.7.2 Comparing profiles

To verify that changes in code really affects performance, one must be able to compare profiles between different runs. Also, solving a performance problem for a specific part of a program may introduce new performance problems in another part.

5.7.3 Controlling measurements remotely

The predefined scenario measurements is relatively simple to use, but the manual configuration option is not as intuitive. A graphical user interface for managing the performance monitor from a host computer would make the manual configuration easier. An application running on a host computer connects to the target node and performs configuration, starting and stopping the sampling process. The application could also be configured to fetch the profile after a run is completed.

5.7.4 Call-stack trace

By including call-stack information in the profile it would be possible to find the execution paths in which the majority of HPC events are sampled. This can perhaps make it easier finding bottlenecks in algorithms that is hard to determine using HPC measurements alone.

5.7.5 Graphical analysis tool

Since CPPMon collects samples in the system scope, a graphical analysis tool should include filter functionality to display samples collected only for the selected programs. The measurement results can be displayed as a histogram, depicting number of collected samples and the responsible function.

5.7.6 Sampler improvements

Setting the event-threshold too low will cause the sampler buffers to overflow. CPPMon cannot extract buffered samples and write them to file fast enough because of disk bandwidth limitations. A fast compression algorithm like RLE applied to the sampler buffers can allow for lower event-thresholds.

The sampling rate is dynamic, and determined by the number of trigger HPC's and the events being measured. Setting the event-threshold too low will

cause the hardware watchdog resetting the system since the HPC interrupt is dominating the CPU. Two possible solutions are: Manually reset the hardware watchdog timer from the HPC interrupt, or define rules for allowed threshold values.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

References

[1] S. Browne et al.
A Portable Programming Interface for Performance Evaluation on Modern Processors. 2000

[2] J.M. Andersson et al.
Continuous Profiling: Where Have All the Cycles Gone? 1997

[3] R. Azimi, M. Stumm, R.W. Wisniewski
Online Performance Analysis by Statistical Sampling of Microprocessor Performance Counters. 2005

[4] W. Korn, P. Teller, G. Castillo
Just How Accurate Are Performance Counters? 2001

[5] *Apple Computer Inc. Computer Hardware Understanding Development (CHUD) tools.*

`http://developer.apple.com/tools/performance} (2006).`

[6] PowerPC 740/PowerPC 750 RISC Microprocessor User's Manual

`http://www-306.ibm.com/chips/techlib/techlib.nsf`
`/products/PowerPC_750_Microprocessor`
`(2006)`

[7] Wikipedia, Out-of-order execution

`http://en.wikipedia.org/wiki/Out_of_Order_execution (2006)`

[8] Enea OSE Systems
OSE Architecture User's Guide

[9] Enea OSE Systems
OSE Documentation volume 1: Kernel

[10] F. Sebek
Instruction Cache Memory Issues in Real-Time System. 2002

[11] M. El Shobaki
On-Chip Monitoring for Non-Intrusive Hardware/Software Observability. 2004

[12] Bryan M. Cantrill, Michael W. Shapiro and Adam H. Leventhal
Dynamic Instrumentation of Production Systems

[13] Dong H. Ahn, Jeffrey S. Vetter
Scalable Analysis Techniques for Microprocessor Performance Counter Metrics

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

6 Appendix: Access configuration

In this section we will explain how the equipment was installed and configured at MDH, user accounts and access rights, and remote access through VPN.

6.1 Mälardalen lab room

A total of 6 machines is located at MDH. Two workstations running Microsoft Windows, a VPN router, a terminal server and two CPP nodes. All development is done on a UNIX workstation in Älvsjö through a form of remote desktop.

6.2 Network

All hosts at MDH is located on the 172.17.252.30/28 network and communicate directly with a gateway in Älvsjö, a firewall is configured initially to deny all access except SSH, HTTP, HTTPS and ICMP ping. Additional ports need to be opened in order to transfer configurations and binaries from the build location to the target nodes. These additional rules needs to be implemented in the Ericsson firewall, preferably on a per-host basis. This is accomplished through placing an order for which services is needed through Ericsson, the actual configuration is done by HP.

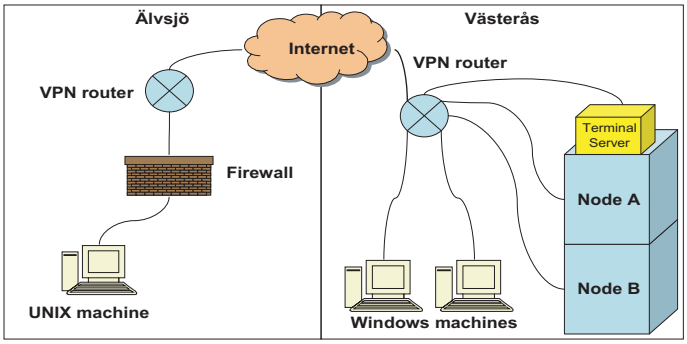

Figure 14: Network configuration

- Initial node configuration can be done manually by transferring configuration and binaries from the build system (workstation) to the node with SFTP. However, it is a slow process and the CoCo tool should be used instead. In order to do this, FTP (21) and Telnet (23) ports must be accessible for outbound traffic from the workstation.
- Additionally, the ports mapped on the terminal server to the serial connections on the nodes must be open for outbound traffic. Each node have one or more serial lines to the terminal server, and the mapped ports start at 10001. In our scenario we have two nodes with two serial links, connected to the first four serial lines of the terminal server. Consequently, ports 10001-10004 need to be opened.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

- Telnet/SSH and FTP access is also required for each node.

6.3 User accounts

The user accounts should be in the *cello* group. Due to security reasons, the accounts that will be used for logging in remotely on the workstation will use a form of fixed config specs to access the ClearCase repositories. These can only be modified by Ericsson staff. The ClearCase access can be requested from the personnel at EAB/UV/Z. Per Börjeson assisted us with our ClearCase configurations.

6.4 Services

A UNIX workstation located at Ericsson serves as development platform. This can be accessed through common SSH, or through a Citrix Metaframe client. The Metaframe client allows for spawning graphical UNIX applications on the client side. The client software used is: *(Citrix Presentation Server Client Packager)* **[?]**

6.5 Terminal server configuration

The IOLAN PLUS terminal server is used to connect to the boards of the CPP nodes when no IP stack is available. This is useful when its necessary to see boot-up messages or when configuring nodes which are in backup-mode. To configure the terminal server, use *telnet* to connect to the terminal server IP, you do not have to specify a port. If a port (10001-10004) is specified the terminal server tries to contact a specific board on one of the nodes connected to the terminal server. A shell should be available, type:

su

followed by the default password for superuser:

iolan

Its possible to view the settings using the *show* command

show gateway

If a default destination already has been added, remove the configuration by issuing:

gateway delete default

To configure the gateway type:

gateway add default [ipaddr] [netmask]

Configure the IP address (and other settings) by typing:

set server

The terminal server will now wait for input for each field, and RETURN is used for confirming any changes made. Use SPACE to scroll through valid options for the different configuration fields.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

6.6 Node configuration

The CoCo configuration file for each node needs to be modified with the network information for the subnet. When the necessary changes have been made, execute the CoCo script with:

```
./coco -group "=>itu_bb_usaal_m4" -format -upload_lm -upload_mo
```

Remember to set your ClearCase view and chmod the coco file to at least 755 first.
If you get an error that the terminal server could not be contacted, make sure that:

- The terminal server have been configured correctly. Try to telnet the terminal server from a local machine.
- You have the required network privileges. Try to access the terminal server from the remote workstation with telnet manually.

We have not been able to determine the full range of ports used by the coco scripts, and the network configuration explained in this document will only allow for configuring the core MP.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

7 Appendix: Design of a time-driven performance monitoring tool

In this section, we will present the design of a non-intrusive performance monitoring application for the CPP-platform/PowerPC 750 processor. By non-intrusive we mean that no target code has to be altered or instrumented.

The design can be broken down into three main parts. A sampler that runs in supervised mode. A daemon process that runs in user-space, serving as an intermediate-layer between the user and the hardware specific sampler. A command-line interpreter that accepts shell commands from the user to configure, start and stop monitoring.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

7.1 System overview

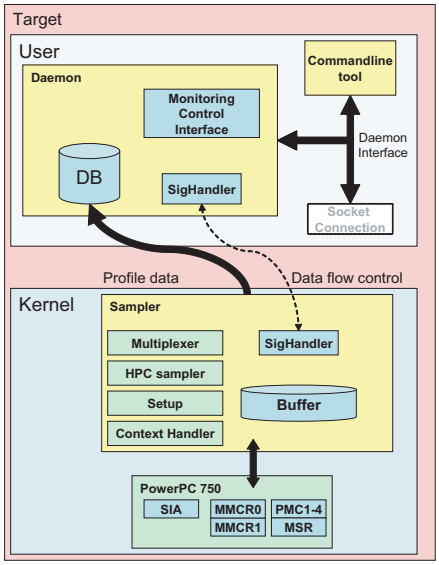

Figure 15: Functional overview of the monitoring system

7.1.1 Disk usage

The disk usage for a sampling period depends on the length of the sampling period, the sampling interval and the number of events.

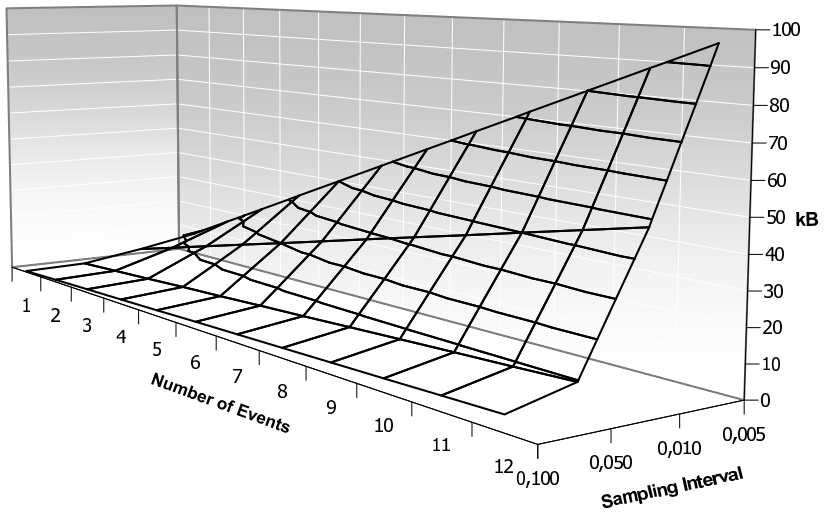

Figure 16: Estimated disk usage when sampling for 10 seconds at varying sample-rates and different number of events

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

7.1.2 CPU overhead

The CPU overhead caused by both the daemon and the sampler are closely related to the sampling rate. A higher sampling rate will result in the interrupt handler running more frequently, filling the sampler data-buffers faster. Consequently, the daemon has to fetch the buffered data in a higher interval.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

7.2 Command Line Tool

7.2.1 Use-Cases and State-Charts

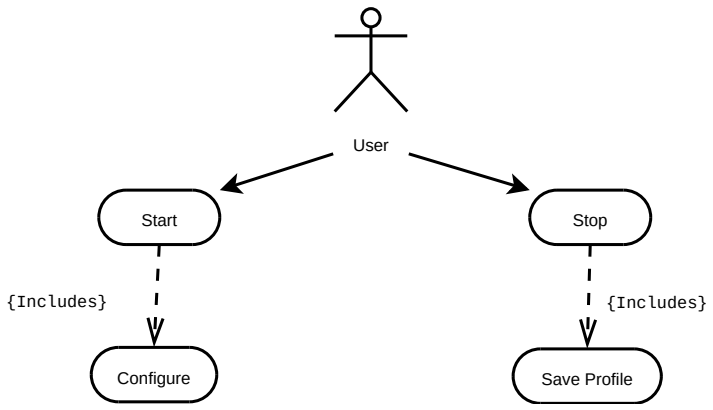

Figure 17: Command line tool use case diagram

- *Start*
  1. User enters command to start monitoring
  2. The daemon is invoked with the selected configuration
     - Exception A: Daemon is already processing another sampling run, report error.
  3. The sampler initializes the PMU and start monitoring
  4. The sampler periodically signals the daemon when sampled data is available.
     - Exception A: Sampling period expired, stop the monitoring process and notify user.
- *Configure*
  1. Command-line tool parses passed parameters or configuration file
     - Exception A: If no parameters was specified, show usage text and exit.
     - Exception B: Invalid parameters or faulty configuration, report error.
  2. The daemon is configured with the given parameters.
- *Stop*
  1. User enters command to halt the monitoring process.
     - Exception A: No sampling session is running, report error.
  2. Daemon notifies the sampler to stop and saves the generated profile.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

- *Save Profile*
  1. Daemon saves the profile on destination given by configuration.
     - Exception A: I/O error, report error.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

7.3 Daemon

The main components in the user-level daemon is the *Monitoring Control Interface (MCI)*, the *Database* and the *SigHandler*.

The MCI is the external interface through which all user-end software communicates with the daemon module. In this project we will implement a command-line interface for controlling the monitoring from the same system. However, the MCI should allow for easy integration of other user-end software, possibly running on a host-machine through a socket connection.
The SigHandler handles the communication between the daemon and the sampler. The daemon configures, starts and stops the sampler through the SigHandler, and the sampler notifies the daemon when data is available. The daemon then extracts the buffered data from sampler.
The Database is a in-memory storage facility where the profile of a sampling run is saved. The Database is updated continuously during the sampling run, and saved to disk once the sampling time expires, or the user halts the process from the command-line tool.

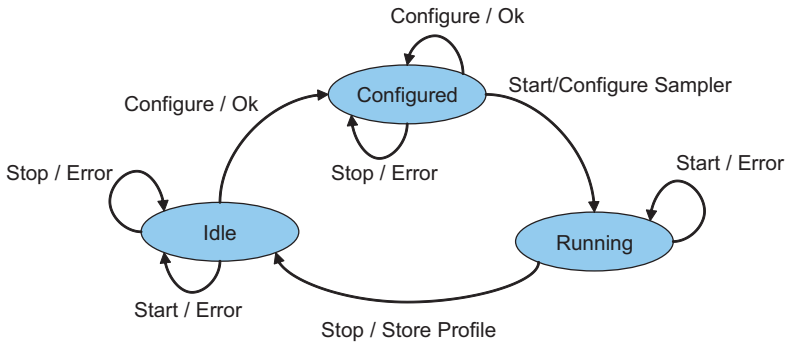

Figure 18: Statechart for the user-level daemon

- *Configure*
  The user-supplied configuration is parsed and a sampler-configuration is constructed. The sampler-configuration is contained within the signal that starts the sampler and consists of the following:

  1. The sampling interval in seconds.
  2. The length of the sampling run in seconds.
  3. All raw events that shall be monitored.
  4. *Optional:* ID of a process to monitor.
     If no ID is specified, the sampler will monitor the whole system.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

The defined events are not directly bound to a specific processor architecture, but rather a collection of event-presets, representing major RISC-like events that can be monitored on most processors with performance monitoring capabilities. This method is adopted from PAPI (section [1]) and should allow replacement of the sampler module for a different CPU-type with little or no reconfiguration of the daemon.

Table 4: Event Presets

ID	**Definition**
EVT_BR	Number of branch instructions
EVT_BR_MP	Number of mispredicted branches
EVT_BR_OK	Number of correctly predicted branches
EVT_CINS	Number of completed instructions
EVT_TOT_CYC	Number of CPU cycles[1]
EVT_L1_IMISS	Number of L1 instruction cache misses
EVT_L1_DMISS	Number of L1 data cache misses
EVT_L2_HIT	Number of data/instruction fetches that hit the L2 cache
EVT_ITLB_MISS	Number of times a fetched instruction was not in the ITLB
EVT_DTLB_MISS	Number of times a fetched operand was not in the DTLB
EVT_FP_INS	Number of completed floating-point instructions
EVT_INT_INS	Number of completed integer instructions
EVT_LS_INS	Number of load/store instructions completed
CMP_L1_DHITRATIO	Indicates L1 Data cache efficiency
CMP_L1_IHITRATIO	Indicates L1 Instruction cache efficiency
CMP_L2_HITRATIO	Indicates L2 cache efficiency
CMP_TLB_DHITRATIO	Indicates data TLB efficiency
CMP_TLB_IHITRATIO	Indicates instruction TLB efficiency

[1] - *One HPC is always dedicated to measure this event.*
It is possible that we will extend this table with specific events for the PowerPC 750 processor during the development.

The sampler is then responsible for mapping the supplied event-presets to the actual bitmasks that is used to initialize the PMU registers (MMCR0, MMCR1 in the PowerPC 750 architecture).

- *Start*
  The SigHandler sends the configuration to the sampler, and the daemon will wait for the sampler to signal that data is available. Different users can use the service that the daemon provides, but only one at a time.

- *Stop*
  If the user requests the monitoring to be halted, the daemon will notify the sampler to stop collecting new samples. The collected samples are

then stored to disk as a profile on the target. The stop signal may also come from the sampler, typically when the sampling period expires. The daemon will then be ready to accept new measuring request from a user.

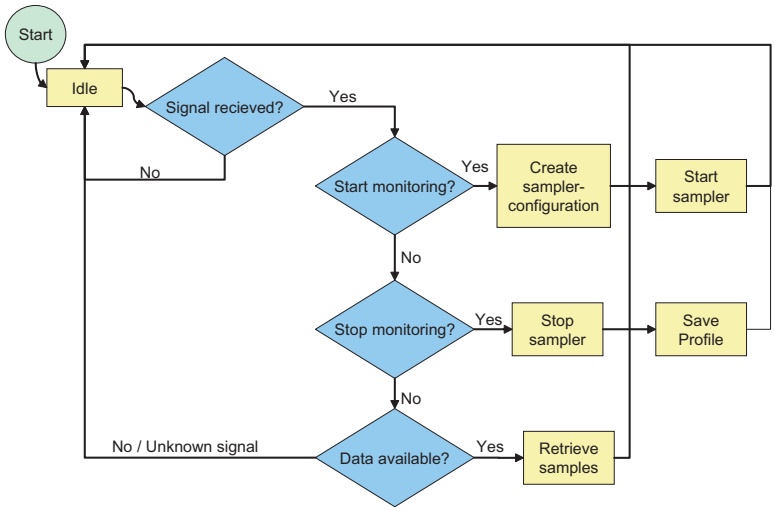

Figure 19: Daemon flowchart

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

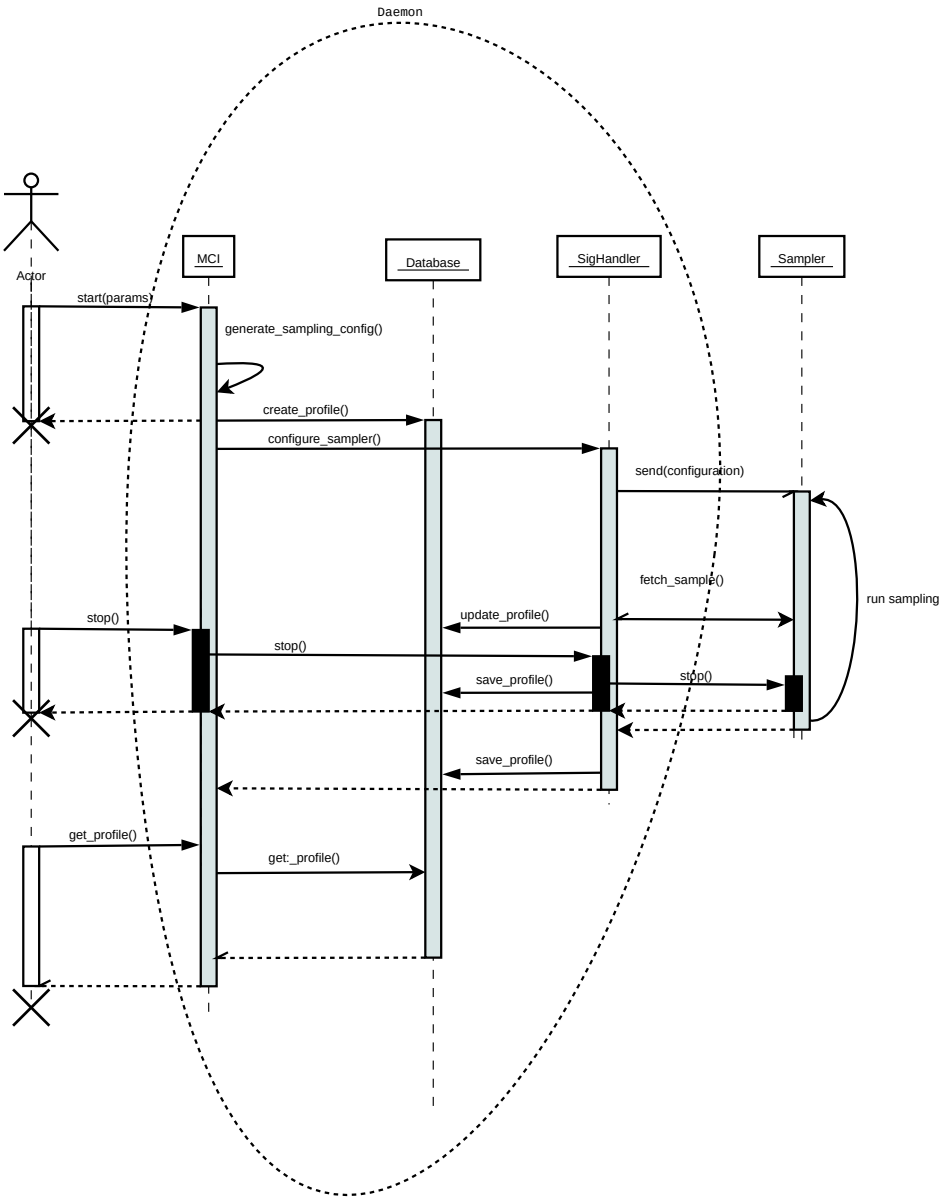

Figure 20: Daemon sequence diagram

7.3.1 Compound statistics

There are several ways of handling compound statistics. One solution is to parse a script containing definitions of the compound statistics to be measured, and extract the necessary raw events that need to be measured and combined according to some formula (given by the script). Compound statistics would be

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

stored directly into the profile instead of the raw events. The user would then not need to bother about the raw events, unless they are explicitly declared in the configuration. Moreover, the size of the profile will be reduced. The drawback of this approach is that it introduces additional workload on the target system, computations and script parsing which easily could have been done on the host machine when presenting the profile.

Another solution is to redirect the responsibility of handling the compound statistics to the user-end application. The daemon will measure the events needed, save the profile and the user-end application would then perform the necessary calculations on the measured events.

We have chosen to leave the calculations required to derive statistics from measured events to the user-end application in order to reduce complexity and load on the target system.

Equations for calculating L1 and L2 cache hit-ratio, memory access density and completed operations per cycle is given in section 3.3.1(PAPI).
Similarily, data TLB hit ratio is defined as:

$$\alpha = 1.0 - (\beta/\theta) \tag{6}$$

Where α spans between $0 - 1$, indicating the ratio of successful DTLB lookups. β represents the number of DTLB misses, θ is the number of load/store instructions completed.
Instruction TLB hit ratio:

$$\alpha = 1.0 - (\gamma/\sigma) \tag{7}$$

Where α spans between $0 - 1$, indicating the ratio of successful ITLB lookups. γ represents the number of ITLB misses, σ indicates the number of completed instructions.

The compound statistics describing TLB efficiency is not tested and verified. The above examples are provided as an example for how raw events can be combined in the user-end application.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

7.4 Sampler

7.4.1 Overview

The sampler is a processor-specific module that reads and configures the performance counters within the CPU. The sampler also implements multiplexing of HPC's and keeps track of in which context a sample is taken. We have chosen a time-driven sampling approach. One of the HPC's in the PowerPC 750 is dedicated to counting CPU cycles, when this counter overflows, all HPC values are read and the results is stored in a buffer.

The sampler is responsible for mapping the requested events into their processor specific counterparts, and the other way around after the events has been sampled. The processor specific events will be represented by a simple bitmask matching the value inserted into the registers that controls the behavior of the HPCs (MMCR0 & MMCR1).

7.4.2 Multiplexing

The sampler is responsible for arranging the events into groups. The events within a group must be simultaneously measurable on the four HPC's available on the PowerPC 750 CPU. Since each HPC can only measure a subset of all available events, the sampler needs to configure the MMCR0 & MMCR1 registers[6] so that no conflicts occur. One HPC is dedicated to count CPU cycles. Upon overflow of this counter the other counters are sampled and reconfigured to measure the next group of events.

Counted events will be linearly interpolated over the whole sampling-round *(R)* which is the number of cycles it takes for all groups to complete. However, it does not make sense to interpolate instruction addresses since the SIA may vary non-linearly during a sampling-round.

Storing the sampled instruction address (SIA) in every group for each overflow would lead to a unnecessarily large profile and possibly inconsistent measurements. Additionally, it would be impossible to determine which instruction address to associate to a compound statistic value. Instead, all groups in a sampling-round *(R)* needs to be assigned the same SIA in order for the sampling round to be consistent.

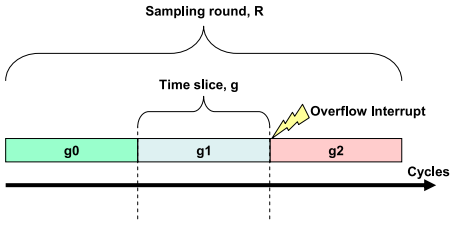

Figure 21: Multiplexing

The interpolation makes it appear as if an event has been sampled throughout the whole sampling-round, however the accuracy will decrease with an

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

increasing number of counted events. Other work [3] has shown that this approach produced acceptable results.

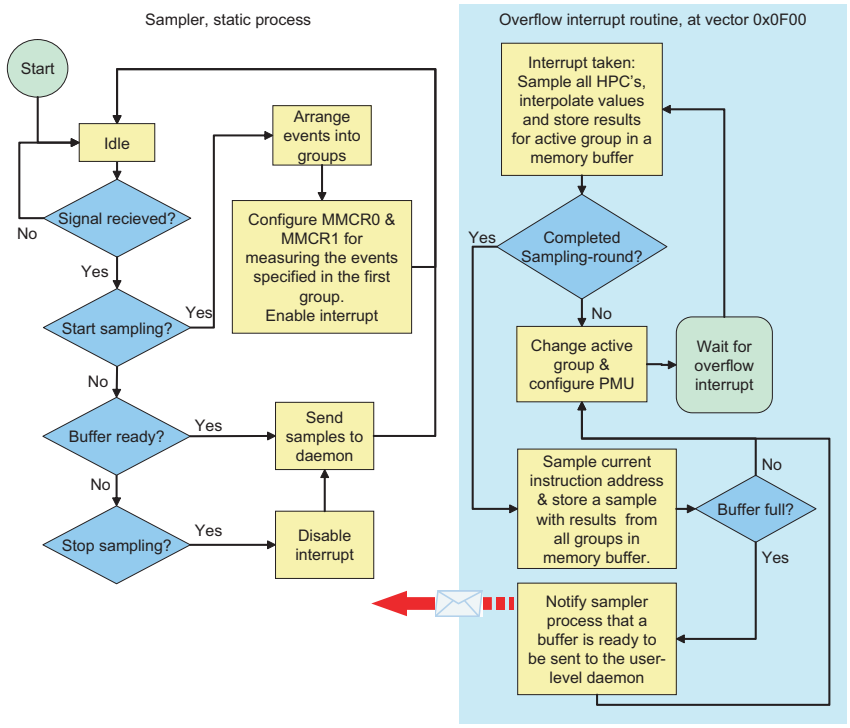

Figure 22: Flow chart for sampler

7.4.3 Sampling Context

Counted hardware events alone provides little help for improving performance of an application. The sampler needs to know in what context each sample is taken. Along with the counted events the sampler stores the ID of the currently running process, the effective address of the instruction executing at the time and in which privilege mode the CPU was in to be able to determine if events occur due to user or kernel-level code. The ID of the process running at the time of the HPC interrupt invocation can be accessed through a global structure in the Cello OSE kernel implementation.

7.4.4 Interface towards daemon

The overflow interrupt handling routine stores samples continuously into a memory area shared with the sampler process. This memory will be split up in segments to allow for double-buffering. When one segment is full, the sampler will signal the daemon to extract the full segment with a custom OSE bioscall.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

During this transfer, the interrupt routine will still be able to fill the other segment with sampled data.

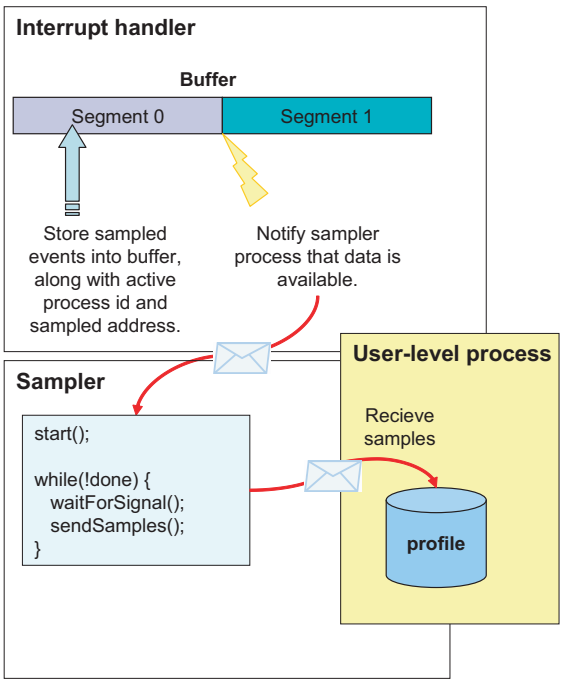

Figure 23: Communication between daemon, sampler and the interrupt handler

Given the sample-rate R, number of events counted in each sample N, the byte-size of each sampled counter S and the size of additional information I that needs to be stored in each sample such as SIA and process ID, the memory bandwidth B that our sampler will use can be calculated with the following formula.

$$B = \frac{S * N + I}{R} \tag{8}$$

B is the number of bytes per second that need to be stored in memory of the sampler-process and at given intervals, transferred to the user-level process (Daemon). The size of each counted event S will match the size of the hardware counter registers (which for a PowerPC750 is 32-bits). As sample-rate increases, the memory usage will increase proportionally as long as the number of events counted remains the same. Naturally, to keep the bandwidth at a constant rate while increasing the number of events being counted the sample-rate will have to be decreased.

To minimize the impact on performance of other running processes, the code for the interrupt handler, needs to be kept small and effective. Therefore, the responsibility of transferring data to the daemon is done by the sampler process (outside the interrupt handling code). Figure 23 shows a simple view of how the dataflow is handled.

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

8 Appendix: Design of an instrumentation performance monitoring tool

In this section we will present the design of an event driven performance monitoring tool focusing on target code instrumentation. By extending the set of available system calls to encompass initialization of HPC's, starting and stopping measurements an application programmer can perform performance analysis during development.

8.1 Kernel extension

The kernel extension consists of a set of performance monitor system calls and facilities for storing samples taken during measurement.

As mentioned in section 2.4, an event driven approach works by waiting for some event to occur, like a L2 cache-hit, and then sample the address of execution within the CPU (and possibly additional useful information). Depending on what processes is currently running, the CPU can generate huge amounts of events for a short period of time. Storing a sample for all these events could cause problems with storage space and performance. One solution to this problem is to set a event-threshold value, indicating the number of events that can occur before a sample is taken. In the PowerPC architecture, the only way to retrieve the address of the instruction executing when an event occur, is to read the SIA register from within an interrupt handler. For these reasons its necessary to let the application programmer limit the scope for where the sampling should take place. This is done by instrumenting the target code with system calls provided by the performance monitoring tool.

8.1.1 Monitoring Context

Code instrumentation alone does not provide any guarantees that samples will contain events generated exclusively for the target process. There are two solutions to this problem, the sampler can query a global structure in the operating system which process was running at the time of interrupt. The accuracy of the samples will decrease with a higher event-threshold, since the HPC's may count events up to threshold-1 from any process running in the system. Another solution is utilizing hardware to mark the target process at the start of a measurement, allowing only events generated exclusively from this process to be counted by the HPC's. This puts the requirement on the OS context switcher to save the process marker bit in the CPU to the process user space.

8.1.2 Storing samples

The Kernel extension will save the generated samples to an internal buffer that can be flushed to disk by the user.

8.2 HPC multiplexing

The multiplexing requirement 3 have been dropped in this design suggestion. The reason for this is the difficulties of interpolating HPC values that arise

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

when each event group can occupy the physical HPC's for different amount of time. The interpolated events cannot be bound to an address of execution, and the SIA of the recently measured event group would have to be copied to all interpolated event groups. There is also an issue of when the groups should be switched. The major benefit of an event driven sampling method is the accuracy of the measurements, using multiplexing would reduce this considerably.

8.3 Public interface

8.3.1 System calls

The interface used by the programmer to control the performance monitoring tool consists of the following functions.

```
/* Clear the performance monitor counters, reset the buffer and
disable PM-interrupts*/
reset_pm();

/* Start measuring the selected 'events', trigger interrupt (store
sample) when 'threshold_value' events have been counted*/
start_pm(unsigned int events, unsigned int threshold_value);

/* Disable counting and interrupts unconditionally*/
stop_pm();

/* Store measured events to file, timestamp will be appended to
filename*/
save_pm(char* file);
```

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54

9 Appendix: CPPMon shell commands

```
DESCRIPTION:
 Start/stop HPC measurement scenario or measure individual events.
        User commands:
        -q                      Force stop measurements
        -s [scenario]           Start scenario measurement:
                                1: L1 cache scenario
                                2: L2 cache scenario
                                3: TLB scenario
                                4: IPC scenario
                                5: Branch scenario
        -e [event] [threshold]  Configure a HPC to measure a specific event, with the specific threshold
        -t [seconds]            Lenght of the profiling run in seconds.
        -h [hpc]                Prints help for HPC 1-4.
        -o [filename]           The location where CPPmon will store the output profile.

        Example: cppmon -e PMC1_CACHE_L1_LOADMISS 0 -t 60 -o test
        See the full documentation for supported events (PowerPC750).
```

Figure 24: CPPMon help section

```
DESCRIPTION: PMU statistics
        Usage:
        smpdiag             Display PMU statistics
        smpdiag stop        Force stop measurements and clear all PMU registers.
```

Figure 25: Smpdiag help section

Erik Hugne
E-Mail: ehe02001@student.mdh.se
Phone: 070-691 14 83

Martin Collberg
E-Mail: martin.collberg@gmail.com
Phone: 076-821 71 54